(2)

A RAND NOTE

An Analyst's Primer for the RAND/ABEL® Programming Language

Paul K. Davis

May 1990

RAND

The research described in this report was sponsored by the Director of Net Assessment, Office of the Secretary of Defense (OSD), under RAND's National Defense Research Institute, an OSD-supported federally funded research and development center, Contract No. MDA903-90-C-0004.

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER N-3042-NA	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) An Analyst's Primer for the RAND/ABEL Programming Language		5. TYPE OF REPORT & PERIOD COVERED interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) P. K. Davis		8. CONTRACT OR GRANT NUMBER(s) MDA903-90-C-0004
9. PERFORMING ORGANIZATION NAME AND ADDRESS RAND 1700 Main Street Santa Monica, CA 90401		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Director of Net Assessment Office of the Secretary of Defense Washington, DC 20301		12. REPORT DATE May 1990 13. NUMBER OF PAGES 43
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No Restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming Languages
Instruction Manuals

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See reverse side

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

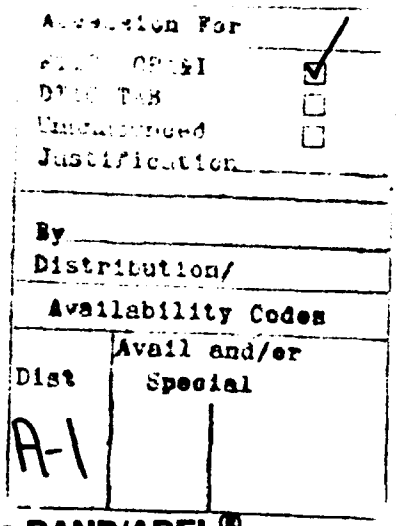
Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

This Note is a primer for analysts wishing to use the RAND/ABEL programming language, a fast, high-level, strongly typed procedural language developed for use in building large and complex knowledge-based simulations in a C/UNIX environment. The primer supplements the comprehensive reference manual by providing a simple introduction and problem sets. The targeted reader is an analyst with subject-area knowledge, modeling capability, and a general understanding of computer programming, but only modest programming skills. After reading this primer and working through the exercises provided, such a person should be able to read and modify substantive logic within RAND/ABEL programs, although sometimes going to the reference manual and depending on more proficient programmers for complex operations or subtle debugging.

A RAND NOTE

N-3042-NA



An Analyst's Primer for the RAND/ABEL® Programming Language

Paul K. Davis

May 1990

Prepared for the
Director of Net Assessment,
Office of the Secretary of Defense

A research publication from
The RAND Strategy Assessment Center

RAND

PREFACE

This Note is part of a continuing project to develop and improve the RAND Strategy Assessment System (RSAS), a project sponsored by the Director of Net Assessment in the Office of the Secretary of Defense. The work was conducted in the RAND Strategy Assessment Center (RSAC), which is part of RAND's National Defense Research Institute (NDRI), a Federally Funded Research and Development Center. Comments and suggestions should be addressed to the author, who directs the RSAC, or to Dr. Bruce W. Bennett, the Associate Director leading the RSAS development project (electronic mail: pdavis@rand.org and bennett@rand.org, respectively). Technical questions about RAND-ABEL should be addressed to Edward Hall or Robert Weissler (edhall@rand.org or weissler@rand.org, respectively).

SUMMARY

This is a primer for analysts wishing to use the RAND-ABEL® programming language, a fast high-level strongly typed procedural language developed for use in building large and complex knowledge-based simulations in a C/UNIX environment. The primer supplements the comprehensive reference manual by providing a simple introduction and problem sets. The targeted reader is an analyst with subject-area knowledge, modeling capability, and a general understanding of computer programming—but only modest programming skills. After reading this primer and working through the exercises provided, such a person should be able to read and modify substantive logic within RAND-ABEL programs, although sometimes going to the reference manual and depending on more proficient programmers for complex operations or subtle debugging.

ACKNOWLEDGMENTS

Colleagues Barry Wilson, Robert Weissler, and Herb Shukiar reviewed this Note in draft and caught a number of mistakes, some stupid and some subtle. They are not to be blamed for those remaining, however.

CONTENTS

FIGURES

I. INTRODUCTION

OBJECTIVES AND APPROACH

The purpose of this primer is to help prospective analyst users of the RAND-ABEL programming language get into applications quickly. The minimum prerequisite is an understanding of basic programming concepts, and preferably at least some modest programming experience.[1] Readers who are or wish to become skilled programmers will later want to master the reference manual,[2] henceforth referred to as RM, which they will use only sparingly in going through this primer and its problem sets (given in the Appendix). Other readers will want instead to depend on programmers for assistance, while they focus on understanding and improving the substantive content of models. Still others will be somewhere in between—*preferring* to do their own modeling and programming, but needing help from professional programmers for relatively complex operations. It is the last two groups to which this primer is largely targeted. As a result, no attempt is made here to be comprehensive.

THE RAND-ABEL® PROGRAMMING LANGUAGE

RAND-ABEL is a fast high-level strongly typed procedural language developed for use in building large and complex knowledge-based simulations.[3] It translates into the C language before execution in a UNIX environment such as that provided by Sun work stations. The language is highly readable and encourages good programming practices. It employs a data dictionary, making self-documentation of programs more straightforward. It includes friendly and powerful new features, notably certain table structures which simplify and clarify knowledge-based models of the "forward-chaining" variety (RAND-ABEL does not support backward-chaining inference of the sort emphasized in PROLOG). The language has been used within RAND for several years, resulting in hundreds of thousands of lines of code in a range of programs that include decision models and algorithmic

---

[1]Readers with a background in such structured languages as Pascal and C will find RAND-ABEL easy to pick up. There are a number of elementary texts for Pascal available that might be useful in elaborating concepts dealt with only briefly here.

[2]Norman Z. Shapiro, H. Edward Hall, Robert H. Anderson, Mark LaCasse, Marrietta S. Gillogly, and Robert Weissler, *The RAND-ABEL Programming Language: Reference Manual*, The RAND Corporation, N-2367-1-NA, December 1988.

[3]For background, see Norman Z. Shapiro, H. Edward Hall, Robert H. Anderson, and Mark LaCasse, *The RAND-ABEL Programming Language: History, Rationale, and Design*, The RAND Corporation, R-3274-NA, August 1985. There have been a number of changes to the language since 1985, but the underlying concepts have remained the same.

calculations. Its optional interpreter permits users to modify code interactively,[4] which has proven invaluable. Current users of RAND-ABEL range from "nonprogrammers," who make focused substantive changes and then depend on magic incantations (commands that work for reasons they don't understand), to professional programmers who use the language for sophisticated applications. The reaction to RAND-ABEL has been quite favorable, although it is more suitable for some applications than others. In its largest application to date, the RAND Strategy Assessment System (RSAS), some of the computationally intensive models are written in the general-purpose language C, while other models are written in RAND-ABEL.[5] Although it is rarely necessary to do so, one can "drop into C" from within a RAND-ABEL program. One might do this, for example, to make use of the extensive C/UNIX library of special functions or to perform efficiently some complex calculation.

RAND-ABEL is available currently in the RSAS, which is a classified system, and in the RAND-ABEL Modeling Platform (RAMP) developed by colleague Edward Hall; RAMP is an unclassified and content-free technology shell consisting of the language and a variety of tools for graphics, logs, and cross referencing. Both the RSAS and the RAMP depend currently on Sun work stations because of system-specific graphics code. RAND is making RAMP available to researchers at no or minimal cost.

STRUCTURE OF THE PRIMER

With this background, Section II of this Note reviews basic programming concepts and terminology, which is important because of the considerable variation in terminology across languages, programmers and modelers. Section III then gives a bottom-up discussion of variables, data types, declarations, operators, and functions. Section IV describes the principal statements of the RAND-ABEL language. Section V suggests how to get started using the language, under the assumption that one already has a working RAND-ABEL program operating on a Sun work station. The Appendix provides problem sets that can significantly help the learning process.

---

[4]That is, one can stop program execution, modify the code, and start up again rather than stopping, changing code, recompiling and starting from the beginning.

[5]Technical readers may wish to see Paul K. Davis and H. Edward Hall, *Overview of System Software in the RAND Strategy Assessment System*, The RAND Corporation, N-2755-NA, December 1988. For examples of RAND-ABEL usage in decision models, see Paul K. Davis, Steven C. Bankes and James Kahan, *A New Methodology for Modeling National Command Level Decisionmaking in War Games and Simulations*, The RAND Corporation, R-3290-NA, July 1986; and William Schwabe and Barry Wilson, *Analytic War Plans: Adaptive Force-Employment Logic in the RAND Strategy Assessment System (RSAS)*, The RAND Corporation, N-3051-NA (forthcoming). For an example of its usage in building combat models, see Patrick Allen and Barry Wilson, *Secondary Land Theater Model*, The RAND Corporation, N-2625-NA, December 1987.

II. BASIC ELEMENTS OF THE LANGUAGE

ELEMENTS OF A PROGRAM: CONCEPTS AND TERMINOLOGY

It is important to give names to each and every element of a computer program. This process is akin to defining parts of speech in English and then diagramming sentences.

Characters are those individual symbols available on the keyboard that are recognized by the RAND-ABEL language—e.g., the letters of the alphabet, the integers 0 through 9, and certain symbols such as $ or *.

Words are packages of characters with no spaces between them. In this definition, 123 is a word, just as much as Edgar or Velocity-4.

Words are either *key words* or *names*. Key words are set aside as part of the language and include commands dictating control (e.g., "If ...Then..." includes the key words **if** and **then**), *operators* such as + and /, and some other items. Names may be names of *variables* (also called *data elements*), *functions*, *inputs* to functions called *arguments* (or, in a more ambiguous usage, *parameters*),[6] and *values* of functions. An *expression* is a word, or a set of words related to or combined with one another via operators, which can be uniquely evaluated (e.g., (4 + 3)).

Computer programs are made up of *comments* and *statements*, which are built up from these words and expressions. Comments are ignored by the computer (i.e., they are not executable). In RAND-ABEL they are enclosed in brackets as in "[This is a comment.]." The executable part of computer programs, the *statements*, are each unambiguous instructions. For example, "Let x be 4." is a statement, as is "Print x."

Functions are made up of statements (if we consider declarations, discussed below, as special cases of statements). The statements may include invoking other functions. Functions themselves can be thought of as black-box segments of the program that do something worth separating off and giving a name to. For example, RAND-ABEL does not itself understand square roots, but one can define a function Square-root that will calculate square roots upon demand. Other functions might consist of numerous

---

[6]Computer scientists use the words variable, data, and parameter quite differently than do most scientists, engineers, mathematicians, analysts, and operations researchers. A modeler's "variables" may be only a subset of the programmer's "variables," and his "data" may be another small subset, the subset of a programmer's "variables" provided in an input statement. Typically, an analyst uses the word "parameter" to mean an item of data that is to be treated as routinely variable. In RSAS documentation, parameters are the subset of input data that can be changed interactively during a run.

statements which together represent something conceptually important, such as suggested by the function name Assess-situation. As mentioned above, functions may have arguments.

Since all of this may seem abstract, consider as a preview of things to come Fig. 2.1, which shows extracts from actual RAND-ABEL code. Note its readability.[7] All comments are enclosed in brackets; the names of things are English-like, except that there are hyphens connecting segments of names. Figure 2.2 shows the same extract annotated to identify the different elements of the code; all key words are indicated in bold. Following custom, we gloss over distinctions between names of things and the things themselves. For example, Fig. 2.2 highlights AFCENT1-deterrence-phase as a function, although one could say that what is highlighted is actually the *name* of the function. In any case, the purpose of AFCENT1-deterrence-phase is to specify orders to certain military forces during a crisis but before conflict begins (what is called here the deterrence phase). The question is what is supposed to happen during the deterrence phase, and under what conditions. We see from the definition that what happens depends on a variable called Point-in-Plan. There are a number of different types of variable; this one is an *array* or what some might call a *vector*. There is a Point-in-Plan value for each of a number of military theaters, of which AFCENT is only one. Hence, the phrase "of AFCENT" is specifying a particular component of the array-type variable. This variable has qualitative values such as Move-to-deterrence. If the If condition is met, then another function will be "performed" (just as, in other languages, one would call a subroutine). Later, the reader will be asked to work through problem sets that include marking up code in this way. First, however, we need to provide more groundwork.

KEY WORDS

True Key Words of the Language

The RAND-ABEL language has many key words (Fig. 2.3), which have specific meaning to the computer and therefore cannot be used as names of variables. The first letter of key words such as Let may be capitalized or not with no change in meaning; the other letters must be in lower case. Numbers are also key words: When standing alone they are recognized as numbers and therefore cannot be names. Thus 145 is a number, while Vel-145 is a name.

---

[7]AFCENT is an acronym for Allied Forces, Central Europe, i.e., a military command; AFCENT1 is one of that command's plans.

```
Define AFCENT1-deterrence-phase:

   If Point-in-plan of AFCENT is at most Move-to-deterrence
   Then Perform AFCENT-deterrence-move.

  While Point-in-Plan of AFCENT is at most Deterrence:
  {
     [ Deploy US and Allied forces as they become available ]

     If (Today is at least C-Day of AFCENT) and
     Authorization of Deployment, AFCENT is Full
     Then Perform AFCENT1-deterrence-deployment-move.
     ...

     If ...
  }
End.
```

Fig.2.1—Illustrative RAND-ABEL computer code

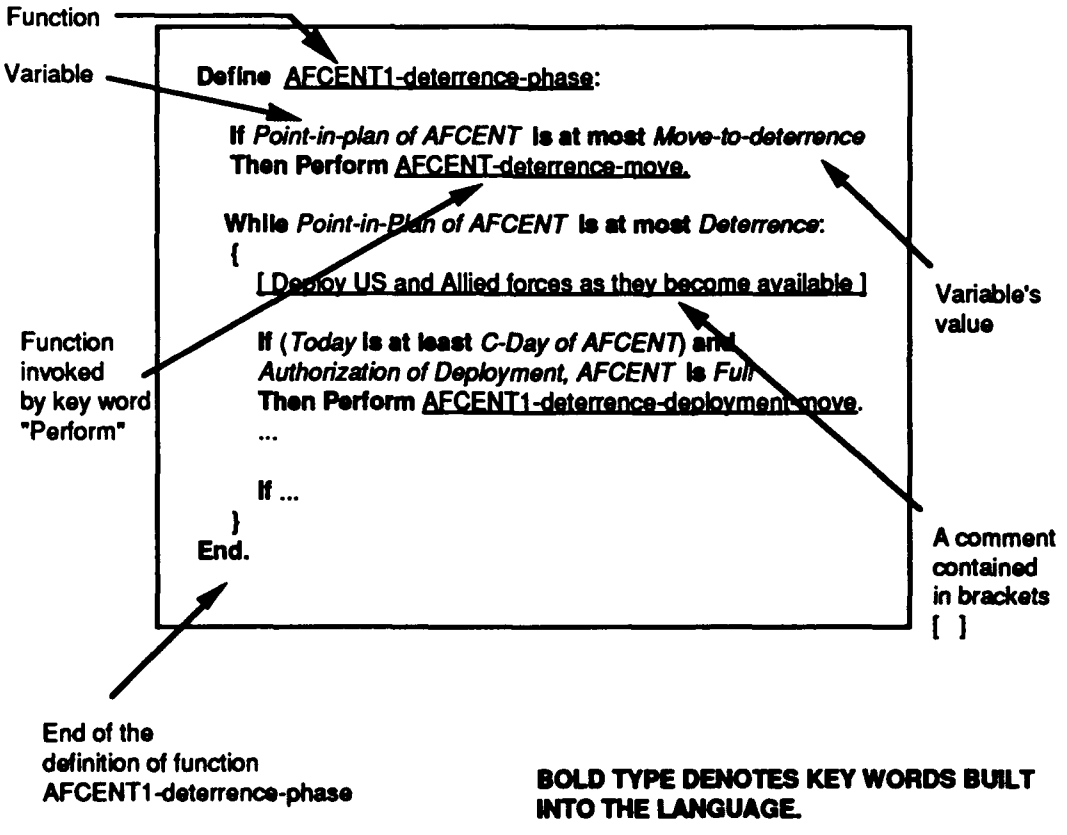


Fig. 2.2—Identifying different elements of RAND-ABEL code

and	End	Let...be	References
are	End Declarations	Log	Report from
are not	Erasable		Reporting
as	Erase	Macro	
Attribute	Evaluate	Make	Self
Author	Everyone	Method	Semi-erasable
	Exit	minus	Status
Begin Declarations		modulo	String
Break	For	Multiply...by	
by	Format		Table
	from	negative	There is
Clone	Function	No	times
Comments		Not	Trace
Concatenated with	Global		
Constant		of	Unerasable
Continue	If...Then	or	Unspecified
	If...Then...Else	Owner	Untrace
Data	Ignore		Use
Declare	In	Parent	using
Declare...	Include	Perform	
by example	Increase..by	plus	Validation
Decrease by	Initialize	Pointer to	Value of
Default	is	Print	
Define	is at least	Prompt	While
#define	is at most		with
Definition	is greater than	Range	Write
Divide...by	is less than	Read	
Divided by	is not	Record	Yes

Fig. 2.3— Key words of the English word variety

Other Reserved Words

For reasons of improving readability, RAND-ABEL has some *noise* or "throw away" words, notably the articles a, an, and the. The computer ignores them. Although these are not true key words, they are unavailable as names. In any given program there may be a number of other words set aside as well, either as noise words or as *aliases* (i.e., *synonyms)* for other words. Aliases are created with a *macro* statement, discussed later.

Operators

Operators may also be considered key words. The most important RAND-ABEL operators are shown in Fig. 2.4. It is preferable to use mathematical symbology where it applies, because that symbology is as much a part of the English language, and more economical than, such prose constructs as "is at least." There are some clever features

Equivalent Forms		Illustrative
Math Form	Prose Form	Usage (a)
+	plus	4 + 3
-	minus	4 - 3
*	times	4 * 3
/	divided by	4 / 3
-	negative	-4
= (b)	is	if x = y then
>	is greater than	x > y
<	is less than	x < y
^		a^2
~=	is not	y ~= x
>=	is at least	y >= x
<=	is at most	y <= x

(a) Spaces are usually necessary before and after operators.
(b) == can be used instead of =.

Fig. 2.4—Most important RAND-ABEL operators

that make the use of operators more English-like. In particular, "is" and "are" may be used interchangeably, which means that one can use proper English with respect to singulars and plurals. There are some other operators discussed in the RM (pp. 20ff.). As in other languages, one can and must use parentheses to resolve ambiguities, but there are also precedence rules to reduce ambiguities. For example, in the expression a^2 * 3, the value of a is squared (^ denotes exponentiation) before multiplying it by 3.[8]

NAMES (IDENTIFIERS)

The various elements of a RAND-ABEL program have names or "identifiers." There is substantial latitude in creating names (RM, pp. 7ff.), as the following examples illustrate:

[8]The exponentiation operator ^ was added since the RM was written.

Escalation-guidance

exchange-ratio

AFCENT-init-forward-defense-order

V2

84flight#-2

Figure 2.5 shows the characters that can be used freely and some characters to be avoided altogether in names. Also, in names capital letters are distinct from small letters (e.g., Abcde is different from abcde).

There are other special rules limiting usage. Although some of them can be violated in certain contexts, the following are good practical rules:

- There can be no spaces within names (absolute rule).
- Avoid key words (e.g., **Let** and stand-alone operators such as +).
- No stand-alone numbers such as 245 can be used as names.
- Avoid -, _, and . as initial or final characters.
- Even though the underscore character _ is valid in RAND-ABEL, don't use it because it causes problems in the Data Editor (not mentioned in the RM).

One cannot necessarily tell from its name what something is in a program (e.g., a variable, function, or comment). Instead, one usually judges that from context. As we saw in Fig. 2.1, function names are accompanied by key words like **Perform**. In an assignment statement (e.g., **Let** ... **be** ...), it must be a variable name that appears to the left of the **be** and the name of a variable value to the right. Skill in identifying elements from context increases as one spends more time with programs.

Valid characters for use in names	Some invalid characters in names
A B ... Z; a b ...z;	" ' \ }] > < >=
1...9	$ * , ^ _
# % + / . &	

Fig. 2.5—Valid and invalid characters in names (identifiers)

VARIABLE TYPES (DATA TYPES)

Variables and Data Elements

Computer scientists often treat "variables" and "data elements" as synonymous. They are the "things" being kept track of in the program. Some of these correspond to real-world objects, some to abstractions describing the real world (e.g., conflict-level), and some to matters of concern to the model or computer program (e.g., the analyst-specified simulation duration measured in simulated days, or the level of detail to be printed out in the simulation's log). Their values may change in the course of the simulation. Operators like + are not variables; nor are key words or functions.

Global and Local Variables

As elaborated later, variables in RAND-ABEL may be defined locally (within a function) or globally. Only global variables' values are consistently stored as the program executes.

Primitive (Built-In) Data Types

The variables or data elements of a RAND-ABEL program fall into nine classes. Six of these are "built in":

Integers (e.g., 1, 4, or 9)

Real numbers, which are better called *decimal numbers* (e.g., 4.3 or, to use scientific notation, 6.02E23, meaning what would be handwritten as 6.02×10^{23}; note the absence of spacing in 6.02E23)

Strings (a series of characters set off by quotation marks, as in "Enemy capitulates")

Boolean or *Logical* (variables that can have values Yes or No)

Process (e.g., RSAS analytic war plans, which are independent coprocesses; see RM for details)

Stream (a data type associated with "pointers" to output files; see RM for details)

As a point of comparison, RAND-ABEL does not currently have built-in complex (imaginary) numbers, double-precision numbers, structures, or certain other data types found in more general-purpose languages such as C.

Constructed Data Types

There are three additional basic variable types: *enumerations*, *arrays*, and *pointers*. These are especially important because one can define an arbitrary number of additional variable types in terms of them. The reason for wanting to do so will become evident later.

Arrays. Just as in the physical sciences the apparent complexity of problems is reduced by introducing the concept of vector or matrix variables (e.g., the equation F = M a substitutes for three equations $F_x = M a_x$, etc.), so also can we simplify programs by using what programmers usually call arrays. An array with one index is equivalent to a *vector*; one with two indices is equivalent to a two-dimensional *matrix*, and so on. In our earlier example, Point-in-plan was a singly indexed array variable. Another might be a variable in the RSAS called Strategic-warning. This is a singly indexed array or vector with as many components as there are theater commands. One of its components might be used in an expression such as "If Strategic-warning of AFCENT > Eur-nuc..." , where the key word **of** highlights that a component of an array is being referenced.

Enumerations. With knowledge-based models, it is important to have variables with qualitative values. RAND-ABEL recognizes a broad class of data types called enumerations, which can have values in a finite ordered set such as {Low, Medium, High}. An example of the data type enumeration is called an enumerated variable.

Because the values of an enumerated variable are in an ordered set, one can write statements such as "If Temperature > Low, Then....", which is then equivalent to the longer statement:

If Temperature **is** Medium **or**
  Temperature **is** High
Then ...

Use of the operators > and < can greatly mitigate combinatorial explosion and increase clarity in complex models. This is not just a programming trick to avoid filling out an enormous decision tree, because in fact we often conceptualize the rules in terms of greater-than or less-than relationships.

All enumerated variables have the special value Unspecified, which is usually denoted - -. Unspecified is an ambiguous concept in that a value may be unspecified because it is substantively unknown (an expression of uncertainty), because it has just not been set yet (perhaps by omission), or because the value doesn't matter substantively

in the particular rule.[9] Global enumerated variables, defined below, have the default value of - -, while local enumerated variables have what programmers call "garbage" as their default value,[10] which means that what the default value is depends on what the computer has recently processed, rather than something consistent and logical. Thus, in using local enumerated variables it is important to initialize explicitly.

Enumerations are defined with statements such as:

Define Enumeration Type-ally: France, Germany, Belgium, Netherlands, UK.

Then, one may have a large number of different variables of the type Type-ally. Such variables have the same range of values (the list of nations shown) and can be combined and compared to one another, despite the "strong typing" defined next.

Strong Typing. RAND-ABEL is "strongly typed," which means that it will not generally permit statements relating or combining variables of different types. This may seem bothersome, because sometimes one wants to do so. However, strongly typed languages are desirable for complex programs, because they greatly improve the likelihood of catching logical or implementational errors early, which is especially useful to nonprofessional programmers. They also reduce ambiguities that might be resolved by the computer in unexpected and mysterious ways (e.g., by arbitrarily assuming the first of two possibilities). In any case, if a modeler conceives a number of enumerated variables that are to be related to one another in rules, then he must define them all to be examples of the same data type. Consider two variables, Color and Mood. Color might have values Red, Green, and Blue, while Mood might have values Angry, Blue, and Querulous. The RAND-ABEL compiler would refuse to accept a statement such as

Let Mood be Color. ,

because the two variables are of different types. However, one could have

Let Johns-mood be Sallys-mood.

---

[9]In the special context of decision tables, as discussed later, one can use ** to mean "any value" in a rule in which one wants to indicate that a particular variable doesn't matter. If one is careful to assure that all variables are assigned values, then - - can be reserved to mean, substantively, "unknown."

[10]The RM is incorrect on the issue of defaults (page 15).

if the two variables were both of type Type-mood with values Angry, Blue, and Querulous. This may seem an annoying complication to those readers familiar with simple programming in BASIC or FORTRAN, which do not have strong typing, but the price paid is small compared to the benefits gained in dealing with complex programs.

Pointers. Pointers can be avoided by beginners to RAND-ABEL, but are useful after one gains some sophistication. To understand them, consider that in normal English one often uses indirection, as in answering "Which size air filter do I put on this car"? The answer might be: "Well, I don't know; go look in the book to see what class this car model falls into and use the corresponding air filter." The second person was answering by "pointing to" where the answer was to be found, noting that it was case dependent. Similarly, in computer programs one may use variables called pointers for indirection. This can speed computational efficiency, improve modularity, and improve program clarity.[11] Pointers are indicated in RSAS code by use of the key words **pointer to**, a synonym **function**, and **value of**.[12]

DECLARATIONS

Syntax

To introduce variables or functions into a RAND-ABEL program one must "declare" their type so that the computer will process them appropriately.[13] This is accomplished by a clever ostensive technique that is especially useful when dealing with enumerated variables. Figure 2.6 illustrates declarations for the most important data types and functions. Bold letters indicate key words. Underlines indicate the evidence the computer uses to infer data type from the declaration. In the first two cases, the evidence is a decimal number and an integer, respectively. In the third case, the key word Perform indicates a function. The next case implies that function2 is a function that produces an integer as output. In the next case, quotes indicate a string variable. In the next, the key word Yes indicates a Boolean variable. The next indicates an enumerated

---

[11]Some of the variables a modeler would identify in his pencil-and-paper work may be represented as pointers in a simulation. In such cases they are not being handled as "state variables." In the simulation, they do not have values to be kept track of. Instead, when the computer processes a rule involving such a variable, it looks to where the pointer points and, typically, computes and uses the value of a function.

[12]There are other synonyms, which are less intuitive and should be avoided. These are **address of** and **occupant of** as synonyms for **pointer to** and **value of**, respectively.

[13]Declarations are not always necessary in other programming languages, but the price paid for this is that the computer *infers* data type, sometimes incorrectly, and one loses the opportunity to detect a great many bugs early (e.g., in languages that don't require declarations, one can mistype a variable name and have the result interpreted as a legitimate new variable).

```
Declare Variable-1: Let Variable-1 be 1.0.

Declare Variable-2: Let Variable-2 be 2.

Declare function-name: Perform function-name.

Declare function2: Let 14 be the report from function2.

Declare Country-name: Let Country-name be "India".

Declare y : Let y be Yes.

Declare tactical-warning: Let tactical-warning be Type-warning.

Declare Strategic-warning: Let Type-warning of AFCENT be Type-warning.
```

Fig. 2.6—Illustrative declarations

variable of type Type-warning, after which we see that Strategic-warning is a singly indexed array, by theater, the component values of which are Type-warning.

Local and Global Variables

As mentioned earlier, local variables are those *used* (referenced via "If variable-name...") or *changed* (via "Let variable-name...") only within a given function. Their declarations must therefore appear within that function. The global variables in RAND-ABEL are variables usable anywhere in the program (except for ownership restrictions described below). Their declarations must appear in something called the Data Dictionary, discussed below. As a stylistic convention rather than as part of the language, RSAS code capitalizes the first letter of global variables, but not local variables.

Ownership

Ownership issues are a bit complex and may be skipped in a first reading. This said, variables and functions in RAND-ABEL have an owner indicated by a name (e.g., Red, Blue, AFCENT, or Edgar). This name may refer to an abstraction such as the RSAS Red Agent representing the Soviet Union, which does not exist per se but which has associated programs called, for example, Ivan or HCFW (an acronym for High Command of Forces West). Alternatively, it may refer to an "access group," i.e., a group of people who are working together on a given portion of software and therefore need to have common access.

The default is Global ownership, in which case the variables or functions can be referenced or changed from anywhere in the program with no special tricks. However, variables or functions owned by, say, Red, may be "hidden" or otherwise "protected." The owner, Red, may specify whether nonowners can *read* or *write* (i.e., see or refer to, or change the value of, respectively). A commonly used option in the RSAS is that if Red owns something, it allows Everyone to read and write it, in which case it can be accessed by, for example, Blue—if the Blue code attaches the prefix "Red's" to fully specify the variable's (or function's) name. Thus, one could write in Blue code that "If Red's variable-name is... Then...".[14] If one doesn't want this kind of cross accessing, which is bad programming practice when overdone, Red can deny read and write access altogether.[15,16]

The ownership feature has many important benefits. At the mundane level, it means that one can use the same name for many different variables or functions, so long as they have different owners (whose names are implicitly part of variables' "full names"). In the RSAS, for example, all the military-command-level models such as AFCENT have mostly the same names for their subordinate functions and key variables (e.g., Point-in-plan, Deterrence-move, FLOT-position-on-axis-3).

Modularity

Existing programs written in RAND-ABEL make extensive use of global variables, which is contrary to what is often described as virtuous design practice with strict modularity. However, by using RAND-ABEL's ownership features one may have a centralized database and still achieve a high degree of modularity, without which it is difficult to build and maintain large and complex programs (see Davis and Hall, 1988, op. cit.). This is a very useful and unusual feature of RAND-ABEL, however unconventional.

---

[14]This can be quite useful in prototype game-structured decision modeling, because it can eliminate the step of having one model "send a message" to another. Instead, each model can monitor some of the other's internal variables.

[15]The ownership restrictions work very much like those associated with human word processing files. An access group, for example, may consist of an analyst and a secretary. There may be a larger group with read-only access, but others can neither read nor change the files.

[16]This is not well documented in the RM, but is discussed in on-line documentation of the Data Dictionary, which is described below.

Data Dictionary

As mentioned above, global variables must be declared in the Data Dictionary, which is a higher-level function drawn upon by the rest of the program. The Data Dictionary is where one finds the precise names and meanings of variables. "Meaning," however, has several connotations that include data type and semantic meaning. A variable's declaration specifies its data type, but its "meaning" must be given in the form of comments. There is no way for the computer truly to "understand" all the variables it is manipulating.

For those able to work on-line with the RSAS or RAMP, the Data Dictionary can be accessed conveniently through the Cross Referencing Tool—e.g., to verify quickly the correct spelling of a variable's name, its data type, its permitted values (and their spelling), and certain other information. It also allows one to view quickly, without visible UNIX operations, the original declaration that should include the comments "explaining the variable's semantic meaning." An example is given in Section V.

FUNCTIONS

Functions are segments of code that may be thought of as black boxes that do something worthy of having a separate name and module. It is good programming practice to break programs down into naturally separable modules so that one can work on one part of the program without worrying about other parts. A function called Square-root might compute square roots, perhaps by calling on a library of functions available to RAND-ABEL as part of the background C-UNIX environment. Another function, Assess-situation, might collect all the statements updating situation in a decisionmaking model.

Functions come in two varieties, those that always return a value and those that never do. The latter correspond to what are called subroutines in FORTRAN.

The syntax for functions in RAND-ABEL can be illustrated as follows.

Functions Returning a Value

For our example, let us assume that H0, g, and T are global variables that have been declared and defined earlier (they represent initial altitude, the gravitational constant, and time). A function Calculate-altitude could be declared and defined as follows:

Declare Calculate-altitude: **Let** 14 **be the report from** Calculate-altitude.
Define Calculate-altitude:
Exit reporting (H0 - 1/2 * g * T^2).
End.

The function could then be used in contexts such as a segment of code making qualitative characterizations of a falling body's altitude:

If the report from Calculate-altitude >= 10000
Then Let Altitude-range **be** High.
Else Let Altitude-range **be** Low.

Functions Not Returning a Value (Subroutines)

Declaration and definition follow the syntax shown below, which assumes that H0, grav, Time, and Current-altitude are valid global variables.

Declare Update-altitude: Perform Update-altitude.
Define Update-altitude:
Let Current-altitude **be** H0 - 1/2 * grav * Time^2.
End.

Usage might then look something like the following, where Altitude-range is another global variable.

Perform Update-altitude.
If Current-altitude >= 10000
Then Let Altitude-range **be** High.
Else Let Altitude-range **be** Low.

Functions Having Arguments (i.e., Taking Parameters)

Functions of either type may depend on one or more arguments. If so, this is specified as part of the declaration. For example, using a variant of the first case above we might have declaration, definition, and usage as follows:

Declare Calculate-altitude:
Let 14 **be the report from** Calculate-altitude **using** 20000 [feet] **as** H0
and 32 [ft per second per second] **as** grav.

```
Define Calculate-altitude:
Exit reporting H0 - 1/2 * grav * Time^2.
End.

If (report from Calculate-altitude using 17000 as H0 and 32 as grav )>
= 10000
Then Let  Altitude-range be High.
Else Let Altitude-range be Low.
```

Here 17000 and 32 are values of arguments. The generalization of this approach is straightforward (see RM, pp. 33ff.).

III. STATEMENTS

As discussed earlier, statements are the executable instructions that constitute programs. RAND-ABEL has the following types of statement, in addition to declarations: (1) Assignments, (2) Conditional executions, (3) Repetitive executions, (4) Table statements, (5) Invoking and exiting from functions, (6) Input and output, (7) Compound, and (8) Null. We shall consider each of these briefly. In some cases there are additional versions of the statement discussed in the reference manual. Here we merely give examples of the proper syntax for standard statements.

COMPOUND STATEMENTS

Although it may seem out of order to do so, let us consider compound statements first so that our examples can be a bit richer in what follows. A compound statement is a series of simple statements collected within braces, as in:

```
If weather is good
Then
{
  Let mood be good.
  Let interest-in-picnic be high.
  Perform Check-picnic-feasibility.
}
```

Each statement within the braces must end with a period.

ASSIGNMENTS

Assignment statements are the bread and butter of computer programs. Denoting key words in bold print, the syntax for the most common assignment statements is as follows.

```
Let Weather be Good.
Let Altitude-range be the report from Calculate-altitude using 17000
as H0 and 32 as grav.
```

Note that the item to the left of **be** must be a variable, while that to the right must be a value, either of a variable or an expression, or, as in the second case, a value returned by a function.[17]

CONDITIONAL AND REPETITIVE EXECUTIONS

If-Then and If-Then-Else Statements

We have already used If-Then statements in our examples. They can be considered primitive concepts here. Just to repeat, however, the syntax for the more general If-Then-Else statement is illustrated by:

```
If report from Calculate-altitude using 17000 as H0 and 32 as grav >=
  10000
Then Let  Altitude-range be High.
Else Let Altitude-range be Low.
```

While Loops

The **While** statement is another workhorse in RAND-ABEL and is used for many statements that might be handled in other languages by **FOR** or **DO** statements. As one example:

```
Let k =3.
While k > 0:
        {
        Print resultsfile k.
        Decrease k by 1.
        }
```

Another example drawn from RSAS work is:

```
While Point-in-plan of AFCENT is Defense

        {
        Perform AFCENT-determine-FLOT.
```

[17]The equal sign (=) may *not* be used in RAND-ABEL for assignments. Thus, "Let Y be 4.0." is valid; "Let Y = 4.0." is not. The equal sign is used in RAND-ABEL *only* for its basic mathematical or logical meaning of equality, as in "If Y = 4.0 Then...".

Perform AFCENT-deterrence-deployment-move.
[Many omitted statements]
If [conditions testing whether Austria is under attack]
Then Perform AFCENT-support-Austria-move.
[Many omitted statements]
If...[details omitted]
Then Let point-in-plan **of** AFCENT **be** Move-to-termination.
}

For Loops

RAND-ABEL's **For** statement is unlike that of most other languages. An example would be:

For alliance-members:
Perform Force-calc using alliance-members as country.

If alliance-members was of type Type-NATO, an enumeration with values equal to the members of NATO, then this statement would have the effect of executing "Perform Force-calc" once for each member of NATO.[18] A useful variant of this syntax is illustrated by:

For alliance-members (US **or** UK **or** France) :
Perform Force-calc.

The parenthetical expression limits execution to the subset of alliance members indicated.

Continue and Break

The key words **Continue** and **Break** can be used in **While** or **For** loops and the general **Table** statement mentioned below. **Continue** means to start the next repetition of the loop. **Break** means to leave the loop altogether, picking up with the next statement. These are both illustrated by the following example in which a test is being conducted on each member of a group, only one part of which is relevant to the test being performed. In this case, it is sufficient that the condition of the test be fulfilled by any one member of the group.

[18]The example given in the RM (page 41) incorrectly omits the colon.

```
For group:
        {
        If item-type of [given item of] group is type-A
        Then Continue.
        [Else if item-type is type-B]
        If position of group > threshold
        Then
                {
                        Let trouble be Yes.
                        Break.
                }
        }
```

TABLE STATEMENTS

Tables are the most innovative single feature of the RAND-ABEL language. They arrange the elements of statements in a familiar and cognitively efficient way—so much so that RAND analysts routinely use precisely the same table structures in brainstorming, model design, and coding. Further, nonprogrammer experts can review the tables directly, with no programmer translations.

One can construct a wide variety of tables using the general concepts described in the reference manual, but two are especially important and will be illustrated here.

Decision Tables

The first and arguably most important table statement is a decision-table version of an If-Then-Else statement. It is illustrated by the decision table at the top of p. 22. [19] Note the periods at the end of the ==== line and at the end of the table. These are required parts of syntax (the RM erroneously omits one of them). The slash (/) separates the inputs from the outputs (i.e., the independent from dependent variables). The operators >, <, and so on must, in decision tables, be butted up against the values that follow them (i.e., no spaces). They greatly reduce the number of lines necessary to complete the decision table and, in many cases, greatly improve the clarity of the logic. Short decision tables can often represent what would be extremely bushy decision trees if

[19]The phrase Decision Table acts like a key word even though it is not one, strictly speaking. Decision is an identifier for a type of table, while Table is a key word. As a practical matter, Decision Table can be regarded as a key word.

Decision Table

Status	Risks	/ Guidance
goals-met	**	termination
>=good	<=medium	no-change
>=good	high	offer-cease-fire
bad	**	offer-cease-fire
very-bad	low	offer-cease-fire
very-bad	>=medium	terminate
--	--	no-change

[End Table].

presented graphically. As illustrated in the problem sets, table outputs can be algebraic expressions or, in effect, pointers to functions. The last line of the table assures that the dependent variable always has a value. This is good programming practice, because local variables without assigned values may be given random values (called garbage) by the computer.

The first line of the table is equivalent to the statement:

If Status **is** goals-met **and**
Risks **are** Unspecified[20]
Then Let Guidance **be** termination.

The second line of the table is equivalent to:

Else
If Status **is at least** good **and**
  Risks **are no more than** medium
Then Let Guidance **be** no-change.

[20]In some older RAND-ABEL code one may see ++ used to indicate "don't care" or "any value"/or variables with numerical values; ** can now be used for either numerical or qualitative values.

Our second example illustrates how equations can be outputs in decision tables. This single table is specifying the equations to be used in 14 different cases.

[Note: The DLR and ER equations are simple approximations of Lanchester equations as defined in CAMPAIGN for CEUR corps-sized LOC battles. Assault-type battles are Hasty-def, Delib-def, Prep-def, and Fortified. The Exploit type battle is currently the same as pursuit.]

Decision Table [LOC axis combat attrition rates; DLR and ER: defender loss rate and exchange rate; FR: force ratio]

type-battle	/ DLR	ER
Cutoff	(.168 * FR / (FR + 2.8))	(6.7 / (FR + 1.8))
Surround	(.168 * FR / (FR + 2.8))	(6.7 / (FR + 1.8))
Delay	(.098 * FR / (FR + 3.6))	(9.0 / (FR + 1.5))
Hasty-def	(.168 * FR / (FR + 2.8))	(6.7 / (FR + 1.8))
Delib-def	(.140 * FR / (FR + 3.0))	(12 / (FR + 2.0))
Prep-def	(.119 * FR / (FR + 3.2))	(16 / (FR + 2.0))
Fortified	(.09 * FR / (FR + 3.6))	(19 / (FR + 1.6))
Meeting	(.21 * FR / (FR + 3.0))	(1.5 / (FR + 0.5))
Static	(.06 * FR / (FR + 6.0))	(3.6 / (FR + 1.8))
Breakthru	0.15	0.33
Exploit	0.06	0.33
Pursuit	0.06	0.33
Pin	0.03	1.3
Advance	0.04	0.75
—	0.001	1.0

[End Table].

Decision tables are the natural format for presenting a large percentage of the rules that constitute typical knowledge-based models such as those found in expert systems. They are much more effective in terms of the reader being able to comprehend the whole. They are also much more efficient, reducing by large multiples the number of lines of code. Most importantly, perhaps, by putting so much of a program's substance in such

tables one maximizes the effectiveness with which analysts and modelers can both review and modify that substance without being expert programmers.[21]

Function Tables

The second widely used type of table is called a function table because it executes a function repetitively. The example shown below assumes a function exists to issue mission orders to the forces on different axes of advance named CEUR-2, CEUR-3, and so on. The table specifies what mission (e.g., "defend, but give ground in delay operations if necessary," as distinct from the "defend" mission which means trying to hold ground) and specifies further the range of area over which the mission applies (the numbers are fictitious). In the example, the first line of the table statement is precisely equivalent to "Perform Axis-mission-order using CEUR-2 as axis..."

Table Axis-mission-order

axis	mission	start-kms	end-kms
=======	========	========	=======.
CEUR-2	Defend-delay	0	109
CEUR-3	Defend-delay	0	100
CEUR-4	Defend-delay	0	134
CEUR-5	Defend-delay	0	126
CEUR-6	Defend	0	40

[End].

Function tables can be highly compact and powerful. One example showing off some of their flexibility is given in the RM (p. 48).

Building Other Table Statements

As described elsewhere (RM, p. 45), the decision table is only one of a great many table statements that can be defined by using the more general **Table** statement built into the RAND-ABEL language. In that statement one specifies as a preamble to the table how each line is to be processed logically. Part of the preamble is called a **macro** and the table is often called a macro table. For example, instead of an If-Then-Else logic as included in the Decision Table statement, one could have a pure If table; in that case,

[21]RAND-ABEL decision tables are an excellent way to accomplish what in some other languages would be done with "case statements."

more lines would be needed in some cases, but each line could be read and understood by itself, without understanding what has already been covered by previous lines. Other possibilities include tables involving If...and...or...Then... (instead of all columns being combined by "and"). A rather clever example of a general Table statement is given at the end of the problem sets in the Appendix.

FUNCTION STATEMENTS

A function is not itself a statement, but "Perform function-name." is a statement, one that invokes the function. Also, there are two special types of statements built into the language for use at the end of function definitions. These are **Exit.** and **Exit reporting ....** As in other languages, **Exit** directs control out of a function during execution; **End** indicates the end of a function's definition.

MACROS

As mentioned earlier, one can establish aliases or synonyms, which have the effect of substituting one string of characters for another. The syntax for this is illustrated by:

#define esc [Global-escalation-guidance].

This defines "esc" as the alias for Global-escalation-guidance. In the unique context of the **#define** statement, items in brackets are not comments. In processing an expression such as "If esc is None...", the actual processing would be of "If Global-escalation-guidance is None...". Macros are commonly used by programmers, but they can cause considerable trouble, since not everyone may remember or recognize them and because they hide important information about what variables are being referenced or changed. Macros can be employed locally or globally, depending on where they are introduced. Finally, it should be noted that words can be declared noise words by a **Declare Ignore** statement, described in the RM.

INPUT AND OUTPUT STATEMENTS

It is beyond the scope of this primer to discuss input/output statements in any detail. Input, in particular, is handled differently in RAND-ABEL than in most languages, at least currently.[22] With respect to output, it is easy in practice to write such

[22]Currently, input is specified by initializing the compiled World Situation Data Set (WSDS) or by writing assignment statements in a file to be executed interpretively (see Section V, which describes use of the interpreter).

statements by copying examples to be found in existing code. Alternatively, one can spend the time to become proficient, which requires some knowledge of UNIX commands or toleration of magic incantations (RM, pp. 52–58). A typical statement generating output in a log file might be:

Log "Since basic-status is good, Let escalation-guidance be" escalation-guidance.

The string in quotes will be printed precisely as written, followed by the current value of the variable "escalation-guidance."

NULL STATEMENTS

Null statements are used occasionally for a variety of reasons, such as setting up some logical distinctions that may be more fully exploited later when the program is given more complex rules. Null statements are of two types: (a) a period or (b) a set of empty curly braces. The syntaxes might be:

If Basic-status is good
Then .
Else If Basic-status is marginal
Then { }
Else
Let Trouble-report **be** Yes.

IV. COPROCESSES

One highly unusual feature of RAND-ABEL is that it allows users to exploit "coprocesses," which are separate computer programs that take turns operating under a main program. In a war game such as the RSAS there can be separate coprocesses for each important player (e.g., for both sides' various theater commanders, as well as for third countries and the sides' political leaderships). Details are beyond the scope of this primer—see, for example, Section 10 of the RM and Davis and Hall (1988)—but it is important for analysts to understand that each of the coprocesses "awakes," executes a portion of its program, "goes to sleep," and later reawakes to continue executing where it left off. A coprocess asks to be put to sleep temporarily by calling an appropriate function, which may also specify when it should be awakened, in terms of either time or condition. An example is as follows:

```
If Today >= D-Day of AFCENT + 3 and
Move-done of AFCENT, Realign-FRG-II-Corps is No
Then Perform AFCENT-Realign-FRG-II-Corps.
[Sleep til tomorrow unless...]
Perform Sleep-to-next-move using the function
Test-nuclear-authorization as planned-wakeup, and
((Today [in days] + 1) * 24) as time-limit [in hours].
[Check for change of phase]
If Authorization of Release, AFCENT is Nuclear...
```

In this example, the program representing the military command AFCENT performs a function AFCENT-Realign-FRG-II-Corps and then goes to sleep either until 24 hours has elapsed or until AFCENT is granted nuclear authorization by higher authority. Upon awakening, the AFCENT program picks up where it left off, first checking to see whether nuclear authorization has been granted. More generally, the sleep statement might be something like: "Perform Sleep-to-next-move using the function Test-wakeup-conditions as planned-wakeup and ((Today + 1) * 24) as time-limit." In this case, the function Test-wakeup-conditions might include a whole series of conditions to be tested regularly. For example, AFCENT might want to wake up if the opponent used nuclear weapons, if there had been high attrition, etc. The typical problem for analysts is to adjust the time of conditions for wakeups.

V. EXPERIMENTING ON-LINE

This section is for readers using a Sun work station with a working program written in RAND-ABEL, either the RAND Strategy Assessment System (RSAS) or programs developed in the generic RAND-ABEL Modeling Platform (RAMP). Such readers can conveniently develop their skills by actually changing and running code. Since this requires at least some understanding of UNIX commands and the RSAS or RAMP environments that goes beyond the intent of this primer, what follows is brief and will be meaningful only to some readers.

The procedure recommended for on-line experimentation is as follows:

1. Have your system administrator or an experienced system user help you to log onto the system, start the RSAS, bring up a "shell window," and move into the directory containing some existing RAND-ABEL code that you would like to read and change as part of your learning experience. You will probably have a prompt sign such as %.

2. Use the "copy" command to make your personal copy of an entire file by typing:

   cp filename testfile.A

   Move the file, testfile.A, into the INT directory under the Run directory. You may need help to do this the first time if you are inexperienced with UNIX.

3. Move to the INT directory and use the editor "e" to open the file you have named testfile.A by typing

   e testfile.A

   and then page through the file to a function that is interesting to you as something to read, modify, and run. Use the Mark and Close commands to delete everything else in the file except the "Owner:" statement at the top of the file.

4. Start by reading the function. Experiment using the Cross Referencing Tool (CR-Tool) that can be activated in the control window. Use the tool to look

up the data type of several variables appearing in the function, the range of values taken on by some enumerated variables if there are any, and their data-dictionary declarations, including possible explanatory comments describing the variables' real "meaning." Figure 5.1 illustrates this for the variable Cohesion-of-NATO.

5. Next, experiment with changing the code. Just use the editor as you would if you were editing a manuscript (except to write RAND-ABEL rather than ordinary English). Remember that you can't just start using new variables because you wish they existed. If you want information that would be input

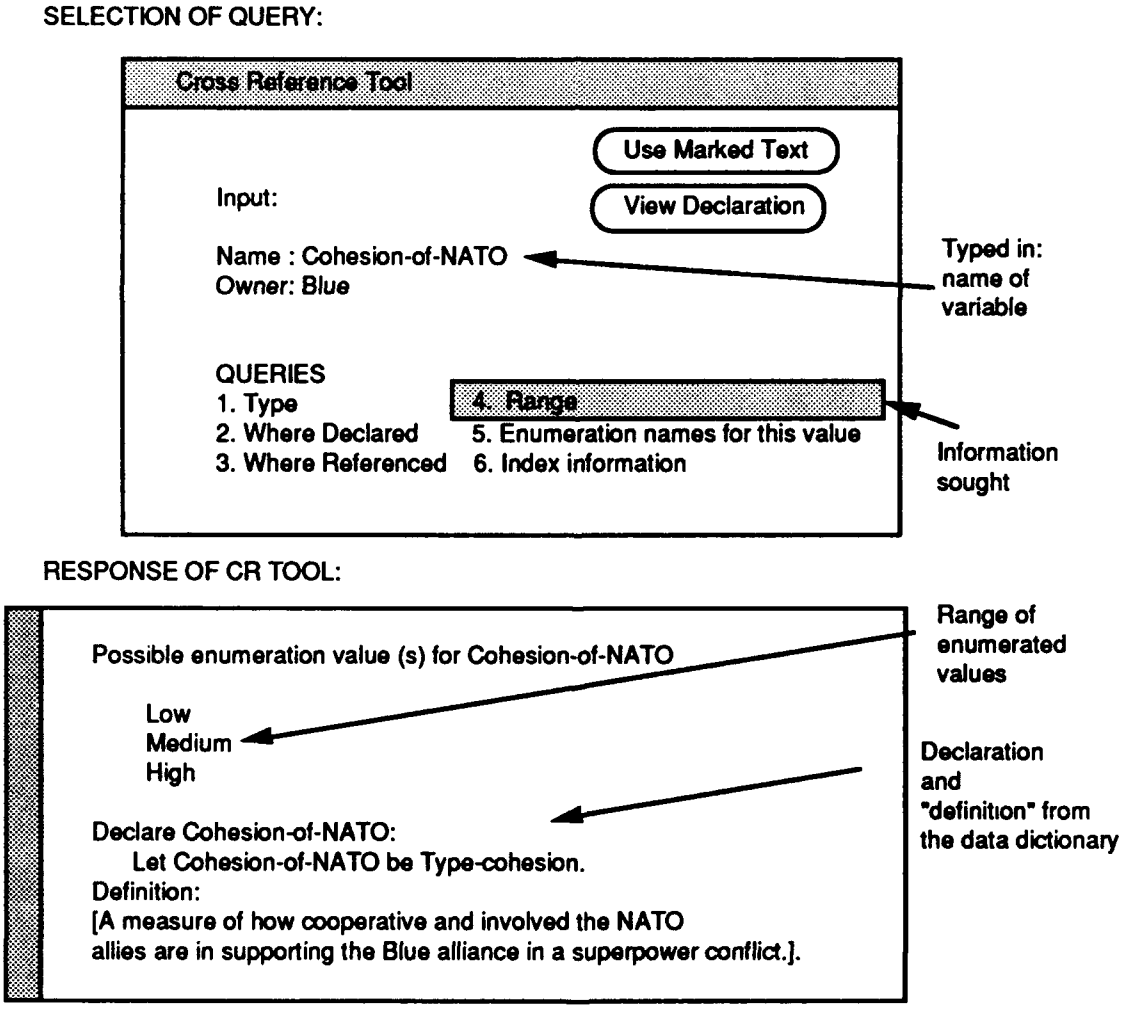


Fig. 5.1—Illustrative use of the RSAS cross-referencing tool (schematic of on-screen display)

for the function, then you must use pre-existing variables available to that function. You may need help to identify your options here. However, so long as you are merely changing logic rather than variables, or using variables that are already being used within the function, you should be all right.

6. When you believe you have a valid change ready to be tested, exit the file.

7. Select the "Interpret now" command from the background menu of the Control Panel and watch the subwindow below the control window to see the results. [23] If you are lucky, it will tell you that the function has now been interpreted. If you are less lucky, there will be error statements. You may have to scroll backward in the window to read them all, although all may be due to a single bug. It is beyond the scope of this primer to teach debugging, or even interpretation of the interpreter's arcane error messages, but from experience we can observe that much can be accomplished by just plunging on. Open the file testfile.A again and look for errors or possible errors. If the error message says there is a bracket missing, that's easy—go look for it. In other cases the message may at least say, in terms of line number, where the computer first got confused. The error may be there or it may be earlier—even much earlier. However, just start looking. Figure 5.2 shows the error message one obtains by trying to declare a new variable called, rather foolishly, "Let." This, of course, is invalid because "Let" is already a key word of the language. Note that there are numerous error messages, even though there is only one error.

8. In practice, working with an experienced user of RAND-ABEL at this stage can be enjoyable and highly efficient. Another practical approach is to browse through existing code, being aware of stylistic tricks as well as

[23]The RAND-ABEL interpreter feature works as follows. When the program is executed, a function appearing in the INT directory with a suffix .A will be compiled and executed in lieu of the original compiled-in version of the function. Execution is perhaps 40 times slower for those functions that are interpreted, but that is usually a very small part of the overall program even if one is doing considerable interactive tinkering.

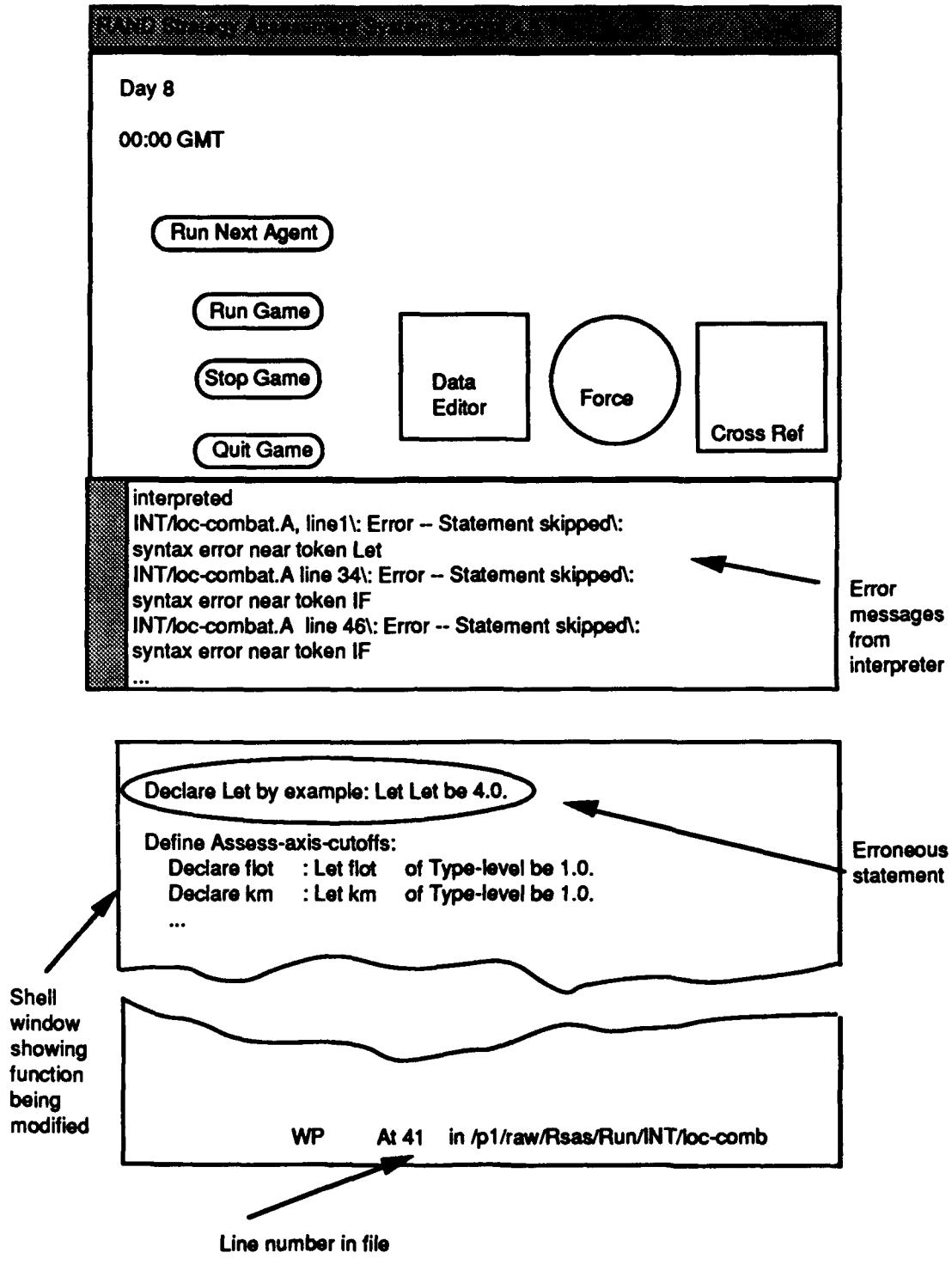


Fig. 5.2—Illustrative error messages while attempting to interpret code (schematic of on-screen display)

syntax. Good examples of RAND-ABEL code can be found using the background walking menus to source code. Or, it may be best to work through some of the baseline-case documentation provided to RSAS users, which walks new users through baseline simulations.

9. For instructions on building programs from the bottom up and, more importantly, on compiling rather than interpreting changes, see Section XI of the RM and ask for assistance as necessary.

VI. NEXT STEPS: BECOMING APPROPRIATELY PROFICIENT

Some readers will wish to become more proficient with RAND-ABEL, but doing so is a matter of degree. One approach is to plunge on with an application, asking for programmer assistance and instruction only when difficulties arise. The result will be, over time, learning what needs to be done for the specific application. Some of what is learned will be utterly mysterious—in the nature of issuing magical incantations. However, that may be the most feasible way to proceed, especially given a minimal background in programming and relatively uncomplex code.

A second approach is to graduate from this primer and some on-line experience to a careful study of the RAND-ABEL reference manual. That is the approach that would be taken by a professional programmer. Or, at least, such a programmer would skim through the reference manual and study the parts he considered unusual and/or nonintuitive. He would then use the reference manual from time to time when he ran into problems.

Much of the complexity of working with RAND-ABEL programs has less to do with RAND-ABEL than with the programs themselves. In particular, most current users of RAND-ABEL are working with the RAND Strategy Assessment System (RSAS), which is a highly sophisticated game-structured knowledge-based simulation of worldwide crisis and conflict. Some of its programs exploit all the special features of RAND-ABEL and contribute new ones specific to their needs. It will generally be possible to read and make changes in individual functions (e.g., functions defining the algorithms for calculating ground-combat attrition or the decision rules for escalation or de-escalation) without requiring programming expertise beyond that covered by the primer or learned quickly from experience. By contrast, much more skill is often needed in changing the control structure of a program, introducing new variables or functions, diagnosing bugs that cut across functions, and so on.

In practice, many RAND analysts who have used RAND-ABEL as part of the RSAS have become fairly skilled programmers, at least in the context of the RSAS. If necessary on a lonely weekend they can do their own debugging or find ways to work around problems. However, it is often more efficient for them to work closely with a professional programmer. The analyst, for example, may sketch out a new model by listing the variables and functions, providing the necessary information for declarations, and writing out the program in pseudo code. Because RAND-ABEL allows constructions

very close to what analysts want to use in any case, the pseudo code can be made almost executable with only a bit of extra effort. The programmer may then take over, make the declarations properly in the right files, change some of the originally proposed names because of pre-existing variables with the same name, correct syntax errors (e.g., insert braces and parentheses), etc. He may then note ways to collapse the logic to be both clearer and more efficient (e.g., through use of While loops or by introducing some logic early on to distinguish among cases that can be treated in separate functions). He then implements this and shows the results to the analyst, who may correct some "corrections" or suggest further improvements. The analyst is often able to run the programs and make significant changes to them interactively with little or no programmer assistance—at least after an iterative period during which the program's limitations are discovered and corrected.

In our experience this approach has been highly successful. The analyst is close to the real program and can determine definitively what it is doing and the degree to which it corresponds to his original model. Further, he can design changes precisely, without having to worry about mistranslations. He can also make some changes himself using the interpreter features. However, efficiency is improved and software quality maintained because of the collaboration with a programmer. In practice, of course, the programmers often become modelers and some analyst/modelers do more programming than they perhaps should.

Appendix

PROBLEM SETS FOR SELF-LEARNING

CHARACTERS, NAMES, AND KEY WORDS

1. The following are examples of what (characters, names, identifiers, operators, or functions)?

   @ 5 * k

2. If your keyboard happened to have the symbol ®, could that be used as a character in an executable RAND-ABEL statement?

3. Which of the following words are valid names (identifiers) for variables or functions in RAND-ABEL? By and large, you should be able to create valid names using your intuition to avoid what "might be dangerous" (e.g., you should suspect that extremely long names requiring a carriage return to print would not work). Occasionally, however, you will make a mistake and receive a bug message when you try to execute the program.

   Snodgrass Init-altitude Macro V0 Gen-con Init.alt V.0. 6 If

   WEATHER-central Ans%$poo Author

4. Since the symbol 6 is recognized by RAND-ABEL as the number six, it cannot be the name of a variable. However, A.6 is a valid name. How does the computer understand that the 6 in A.6 is not a number but merely a character that is part of a name?

5. Which of the following are invalid as a key word?

   Report from report from REPORT FROM report From

5. Since both End and end mean the same in a RAND-ABEL program, do Edgar and edgar?

NUMBERS , ALGEBRA, AND LOGIC

1. Which of the following are valid ways to express Avogadro's number in RAND-ABEL code?

   6.02 E 23 6.02 * 10**23 6.02 E23 6.02E23 6.02 * 10^23

2. Re-express each of the following using math notation valid in RAND-ABEL:

If Y is 6 and
  X is (a divided by b)
Then...

If Y is greater than 3
Then Let X be Yes.

If Y is not 3
Then Let X be Yes.

3. Which of the following is intended to be equal to 1 in RAND-ABEL? Re-express all of them using parentheses or spaces to assure that they are both readable and unambiguous to the computer.

   2^3 /8 (2^3)/8 (2)^3/8 ((2)^3)/8

4. Which of the following is an invalid RAND-ABEL expression, and why? There may be more than one error per item.

   (((2+3)/4 + 5)

   (2**3)*((4+5)/4.5)**2)

   If weather is good
   Then
     [
       Let situation be good and Let Call-for-Party be Yes.
     ]

5. Why is the following invalid in RAND-ABEL? Write a correct version with the same intended meaning.

   If Box-status is empty and
     Supply-status is little or none
   Then Let Situation be poor.

RECOGNIZING THE ELEMENTS OF A PROGRAM

1. Mark up Figs. A.1–A.2 in the same manner as Fig. 2.2 of the text. Remember that every item must be one of the following:

   - Key word (treat noise words and operator symbols as key words)
   - Variable
   - Variable value
   - Function
   - Argument of (or parameter of) a function
   - Value of a function
   - Comment

```
If the Temperament of the Actor is Moderately-reliable
  and the Effectiveness of the Actor is Low
  and the Conflict-location-status of the Actor is None
Then
{
    Let the Side of the Actor be White.
    Let the Cooperation of the Actor be Normal.
    Let the Preparedness of the Actor be Normal.
    ...
}
Else If the Temperament of the Actor is Reliable...
```

Fig. A.1—Illustrative RAND-ABEL code (1)

```
If Current-situation is Eur-demo-tac-nuc
Then
Decision Table
                                    Escalation-
Basic-status        Risks       /   guidance
============    ========    /     ============.
goals-met           --              Eur-term
progress-good       low             Eur-demo-tac-nuc
progress-marginal   low             Eur-gen-tac-nuc
progress-good       marginal        Eur-demo-tac-nuc
progress-marginal   marginal        Eur-gen-tac-nuc
--                  --              Eur-demo-tac-nuc
[End Table].
```

Fig A.2—Illustrative RAND-ABEL code (2)

2. Go back to Figs. A.1-A.2 and identify examples of statements, compound statements, and expressions.

RECOGNIZING OR ESTABLISHING DATA TYPES

1. Consider the segment of code below and circle each of the variables and indicate its data type:

```
If weather is good and
        temperature >= 68 and
        humidity <= 50.0 [%] and
        location-of-friends is local
Then
{
        Let trip-appropriate be Yes.
        Print "Trip is possible.".
}
Else
 {
        Let trip-appropriate be No.
        Print "Trip is not possible. #$!!@#".
 }.
```

2. In the decision table below, what kind of variable is each-arena? What are its values? What about referee-function? Explain what the decision table is doing. (Hint: this code is being used like a "case statement.") Suppose you were on-line with the RSAS and wanted to use the cross referencing tool to help. If you were unable to find each-arena or referee-function using the tool, what would the likely explanation be? Where would you then look?

```
Decision Table

 each-arena          /     referee-function
=============        /     ======================.
NEUR                       (function Norway-referee)
TF-Baltic                  (function Zealand-referee)
AG-Balkan                  (function Greece-referee)
AG-Turkey                  (function Turkey-referee)
 ...
CEUR                       (function CEUR-referee)
--                         (function Do-nothing)

[End Table].
```

3. For those who have the RSAS or RAMP and are able to experiment on line, find convenient sections of code and use the Cross Referencing Tool to verify your assessments of data type, as well as to find their ranges of values.

DECLARATIONS

1. Suppose you have constructed a paper-and-pencil model for calculating the height of a falling body. You have written it as:

$$H = H_0 - 1/2\ g\ t^2$$

Upon trying to represent this in a program, however, you find that current time is represented by the variable Time and that there is no concept of the gravitational constant within the program. The body's initial height is represented by a pre-existing variable Alt, which applied at a time Time-init, but Alt is measured in feet from the surface of the earth and you want to have height be represented in nautical miles. You only need to use the variable height "locally," i.e., in a single function where it is needed as part of another calculation involving the expected drag force. Write code that would appear entirely within this function to declare appropriate local variables and implement the formulas listed above.

2. (A challenging problem that will require using the reference manual). Suppose you must declare a long list of variables of various types. Sketch out a table statement that would prove convenient to this purpose—i.e., design the table you would like to fill in. Subsequently, see if you can define that table using the general Table mechanism described in the reference manual.

STATEMENTS

1. Write a decision table equivalent to the following, which a modeler might have scrawled out in longhand.

```
If air-control is attacker and
  strategic-surprise-of-defender is High
Then Let local-surprise-of-defender be High.
Else If air-control is attacker and
  strategic-surprise-of-defender is Medium
Then Let local-surprise-of-defender be High.
Else If air-control is attacker and
  strategic-surprise-of-defender is Low
Then Let local-surprise-of-defender be Medium.
```

Use examples in the reference manual to help construct the column headings neatly, since some of the variable names are quite long.

2. Consider the decision table below. Note that some of its output columns involve formulas. What is the prose version of the second line of the table? Here FR stands for force ratio, ER for exchange ratio (attacker losses divided by defender losses), and DLR is the defender's daily loss rate, as a fraction of defender forces. What could be used instead of ++ to mean "any value"?

Decision Table [point loss rates]

target	battle	FR	/ DLR	ER
Landchoke	Breakthru	++	1.0	0.50
Landchoke	Hasty-def	++	(.25 * FR /(FR + 2.80))	(7 / (FR + 1.8))
Airfield	Hasty-def	++	(.34 * FR / (FR + 2.80))	(6 / (FR + 1.8))

...

[End Table].

3. Consider the decision table below, where x and y are variables with values Low, Medium, and Large; and z is a variable with values Bad, Fair, and Good. Look at the last line of the table. Does it imply that so long as y is Low, z is Bad? Why or why not? Rewrite the table to have one line for each logical case (i.e., rewrite without using the operators > and <).

Decision table

x	y	/	z
==	==	/	=.
>Low	>Low		Good
**	>Low		Fair
**	Low		Bad

[End].

4. The following illustrates the use of the general Table statement, which allows the user to define the way in which the various column variables are processed. It tests to see whether named air force units have already been deployed to their desired destination; if not, it directs that they be so deployed. Study this example and then answer the following questions: (a) What is executed next if the condition leading to the "Continue" is met?; (b) How could the table statement be modified so that the deployment order is given only after a certain time in the simulation? [Hint: assume Today-in-days is a global variable.]

```
Define AFCENT-move-planes:

  Table
  {
        Declare index:  Let index be 1.
        Declare axis:   Let axis be Type-axis-overlay.
        Declare unit:   Let unit be "string".
        Declare owner: Let owner be Type-country.
        Declare to-region: Let to-region be Type-region.

        If AFCENT-squadron-moved of index is Yes
        Then Continue.

        Perform Deploy-air-order using unit as unit-name,
              owner as owner, and to-region as to-region.
        Let AFCENT-squadron-moved of index be Yes.
        Log-decision "  Moving " unit " to" to-region.
  }
```

index	axis	unit	owner	to-region
1	CEUR-10	"37th-FBW"	FRG	France
2	CEUR-6	"41st-LAW"	FRG	Denmark

[this could go on for dozens of lines]
[End Table].

INDEX